Software Engineering Group
Department of Computer Science
Nanjing University
http://seg.nju.edu.cn

Technical Report No. NJU-SEG-2016-IC-008

2016-IC-008

Automatic Invariant Synthesis for Arrays in Simple Programs

Bin Li, Zhenhao Tang, Juan Zhai, Jianhua Zhao

International Conference on Software Quality, Reliability and Security 2016

Automatic Invariant Synthesis for Arrays in Simple Programs

LI Bin, TANG Zhenhao, ZHAI Juan, ZHAO Jianhua
State Key Laboratory of Novel Software Technology
Dept. of Computer Sci. and Tech. Nanjing University
Email: {hsslb,tangzh,zhaijuan}@seg.nju.edu.cn, zhaojh@nju.edu.cn

***Abstract*—This paper proposes a way of using abstract interpretation for discovering properties about array contents in programs which manipulate arrays by sequential traversal. The method summarizes an array property as a universally quantified property. It directly treats invariant properties (including universally quantified formulas and atomic formulas) as abstract domains. Our method is sound and converges in finite time, and it is flexible. The method has been used to automatically discover nontrivial invariants for several examples. In particular, the method can represent and process multidimensional array properties.**

I. Introduction

An array is a simple and efficient data structure that is heavily used. In many cases, to verify the correctness of programs that use arrays an analysis needs to be able to discover properties of array elements.

Static reasoning about the behaviors of such programs is a challenging problem. Firstly, array indexing induces complex semantics, in particular the possibility of aliasing. Moreover, an array may represent a large or unbounded number of variables, since the size of the array can be large or unknown. Reasoning about the behaviors of array programs always requires the use of universal quantifiers.

Invariant synthesis for programs manipulating arrays with unbounded data has drawn wide attention in the academic field. Predicate abstraction [10], [21] uses some easy syntactic heuristics to derive the predicates used for the abstraction. Moreover, counter-example guided refinement [5] and Craig interpolants [17] propose a significant improvement. Array smashing approach [7] means all the cells of an array are subsumed by one variable, of the same type as the cells. Array Partitioning [11], [13], [15] proposes a significant improvement by partitioning the index domain (say, [1...n]) into several symbolic intervals and associating with each subarray a summary auxiliary variable. Under-approximations and Templates method [14] is extremely powerful yet expensive. It requires: (i) the user to provide templates for the array invariants; and (ii) the abstract domain to perform under-approximations for the index variable. Theorem prover-based method [19] uses a saturation theorem prover to generate loop invariants.

In this paper, we propose a way of discovering universally quantified properties about array contents in some restricted cases by using abstract interpretation framework [18]. We restrict analysis to "simple programs", which manipulate arrays only by sequential traversal: typically, increasing (or decreasing) their index by a fixed constant in loop body, and accessing arrays by simple expressions (constant translations) of the loop index. A lot of programs or code snippets fall into this category. We focus on $<$, $\leq$, $=$, $\neq$, $>$ and $\geq$ relationships between traversed array cells and scalars, or relationships between traversed array cells. The followings are some examples of properties. The formal exposition shall be given in the subsequent sections.

$\forall k.k \in [0, 2, i), A[k] > x$

$\forall k.k \in [n-1, -1, i), A[k] = B[k]$

$\forall k_1.k_1 \in [0, 1, i), \forall k_2.k_2 \in [0, 2, j), A[k_1][k_2] = B[k_1][k_2]$

$\forall k.k \in [0, 1, i), A[k][k] = B[k]$

In the above, [Init,c,final) denotes a set,

$\{Init + n * c | n >= 0 \wedge Init + n * c < final\}$ *if c is positive*,

$\{Init + n * c | n >= 0 \wedge Init + n * c > final\}$ *if c is negative*.

The contributions of our work include:

- Our main contribution is to propose a fully automatic method to discover properties about array contents. In particular, our method can represent and process multidimensional array properties. We provide soundness proofs and convergence proofs for the method.
- We describe a framework of invariant properties (including universally quantified formulas and atomic formulas) abstract domain. The invariant properties abstract domains are highly expressive and flexible.

II. Language and preliminaries

We first define the "simple programs" in which we formalize our technique.

A. Simple Programs

The "simple programs" means manipulating arrays only by sequential traversal: typically, for loops incrementing (or decrementing) their index by a fixed constant, and accessing arrays by simple expressions (constant translations) of the loop index. Different from

978-1-5090-4127-5/16 $31.00 © 2016 IEEE
DOI 10.1109/QRS.2016.23

the similar definition in [15], we allow programs to increase (or decrease) their index by a fixed constant (instead of just 1), and allow multidimensional arrays in programs. We show several examples of such "simple programs", which will be used throughout the paper. Note that, the loop index can be increased (or decreased) multiple times at each iteration, see Program 4.

Program 1. arrayPartCopy

```
unsigned size;
int A[size],B[size];
i=0;
while (i<size) {
    A[i]=B[i];
    i=i+2;
}
```

Program 2. twoDimArrayCheck

```
unsigned col,row;
int A[row][col];
for (int i=0;i<row;i++)
    for (int j=0;j<col;j++)
        if (A[i][j]==0)
            return true;
return false;
```

Program 3. another twoDimArrayCheck

```
unsigned col,row;
int A[row][col];
bool flag=false;
for (int i=0;i<row;i++) {
    for (int j=0;j<col;j++) {
        if (A[i][j]==0) {
            flag=true;
            break;
        }
    }
    if (flag==true) {
        break;
    }
}
return flag;
```

Program 4. Find: segmentation phase of the QuickSort

```
unsigned size;
int A[size];
int x = A[0] ;
int i = 1;
int j =size−1;
while (i<=j) {
    if (A[i] < x) {
        A[i−1] = A[i];
        i = i + 1;
    }
    else{
        while (j>=i&&A[j]>=x) {
            j = j−1;
        }
        if (j > i) {
            A[i−1] = A[j];
            A[j] = A[i];
            i = i + 1 ;
            j = j−1;
        }
    }
}
A[i−1] = x ;
```

The following gives syntax of statements:

$$
\begin{aligned}
\mathit{statement} ::= {} & \mathit{lhs} = \mathit{expr} \\
& \mid \mathit{index} = \mathit{init} \\
& \mid \mathit{index} = \mathit{index}(\text{`}+\text{'} \mid \text{`}-\text{'})\mathit{constant} \\
& \mid \textbf{while}(\mathit{cond})\ \mathit{statement} \\
& \mid \textbf{if}(\mathit{cond})\ \mathit{statement}\ \textbf{else}\ \mathit{statement} \\
\mathit{lhs} ::= {} & \mathit{Cvars} \mid \mathit{Array}; \\
\mathit{index} ::= {} & \mathit{Indices}; \\
\mathit{init} ::= {} & \mathit{expr}; \\
\mathit{constant} ::= {} & \mathit{INT}; \\
\mathit{Cvars} ::= {} & \text{`}x\text{'} \mid \text{`}y\text{'} \cdots; \\
\mathit{Array} ::= {} & (\text{`}A\text{'} \mid \text{`}B\text{'} \cdots)([\mathit{Iexp}])+; \\
\mathit{Iexp} ::= {} & \mathit{constant} \mid \mathit{Indices} \mid \mathit{Indices}\ (\text{`}+\text{'} \mid \text{`}-\text{'})\ \mathit{INT}; \\
\mathit{Indices} ::= {} & \text{`}i\text{'} \mid \text{`}j\text{'} \cdots;\ (\mathit{Indices} \cap \mathit{Cvars} = \emptyset) \\
\mathit{cond} ::= {} & \mathit{expr}\ (\|\ \mathit{expr})*;
\end{aligned}
$$

Assignments can assign values to scalars, array elements or *indexes*. Moreover, assignments to *index* are divided into two kinds: *index* initialization statement "*index* = *init*" and *index* update statement "*index* = *index (+|-) constant*". The *index* does not occur in the expression *init*. In *index* update statement "*index* = *index (+|-) constant*", the first *index* is same as the second *index*. Index expressions *Iexp* are restricted to constants, indexes or sums of an index and a constant. Conditions are conjunctions and/or disjunctions of atomic conditions.

B. Properties

The Properties used in the analysis are defined as:

$$
\begin{aligned}
\mathit{prop} ::= {} & \mathit{andprop}(\vee\ \mathit{andprop})*; \\
\mathit{andprop} ::= {} & \mathit{atomprop}(\wedge\ \mathit{atomprop})*; \\
\mathit{atomprop} ::= {} & \mathit{simpleProp} \mid \mathit{forAllProp} \mid \text{`}(\text{'}\mathit{prop}\text{`})\text{'}; \\
\mathit{simpleProp} ::= {} & \mathit{expr}\ \mathit{op}\ \mathit{expr}; \\
\mathit{forAllProp} ::= {} & \forall \mathit{ID}\ \text{`}.\text{'}\ \mathit{ID} \in \mathit{interval}\ \text{`},\text{'}\ (\mathit{atomprop} \mid \phi); \\
\mathit{op} ::= {} & > \mid \geq \mid < \mid \leq \mid = \mid \neq; \\
\mathit{interval} ::= {} & (\text{`}(\text{'} \mid \text{`}[\text{'})\ \mathit{expr}_1\ \text{`},\text{'}\ \mathit{constant}\ \text{`},\text{'}\ \mathit{expr}_2\ (\text{`})\text{'} \mid \text{`}]\text{'});
\end{aligned}
$$

Properties can be conjunction properties or disjunction properties. Atomic properties can be *simple properties*, *universally quantified properties*. The *simple properties* are $<$, $\leq$, $=$, $\neq$, $>$ and $\geq$ relationships between expressions. We use ϕ to denote a special property, which is used in the analytic process. The detailed descriptions of ϕ shall be given in the subsequent sections.

The *interval* starts from $expr_1$ (included or excluded) to the $expr_2$ (included or excluded), and the difference between the consecutive terms is constant *constant*. If *constant* is a positive number, and $expr_2$ is less than $expr_1$, the interval is empty and vice versa.

Table I
THE MEMORY SCOPE FOR BASIC FORMS OF TERMS

terms	memory scope
a constant **c**	$\emptyset$
a variable **v**	{&v}
$e_1[e_2]$	$\{\&e_1[e_2]\} \cup \mathbb{M}(e_1) \cup \mathbb{M}(e_2)$

Table II
THE MEMORY SCOPE FOR PROPERTIES

properties	memory scope
e_1 bop e_2	$\mathbb{M}(e_1) \cup \mathbb{M}(e_2)$
$\forall x.x \in [e_1, c, e_2), \phi$	$\mathbb{M}(e_1) \cup \mathbb{M}(e_2)$
$\forall x.x \in [e_1, c, e_2), p$	$\mathbb{M}(e_1) \cup \mathbb{M}(e_2) \cup \{\mathbb{M}(p) \mid x \in [e_1, c, e_2)\}$

C. Memory Scopes of Properties

The concept of memory scopes comes from [25].

For a property P, the value of P depends only on the contents stored in a finite set of memory units. We use $\mathbb{M}(P)$ to denote the memory scope of P. $\mathbb{M}(P)$ can be constructed syntactically.

For an expression e, if e is of the form $f(e_1, \cdots, e_n)$, where f is an algebraic operator (e.g. $+, -, ...$) or a boolean operator, the memory scope $\mathbb{M}(e)$ is $\mathbb{M}(e_1) \cup \mathbb{M}(e_2) \cdots \cup \mathbb{M}(e_n)$. Table I shows the memory scopes for basic forms of expressions.

Now we can define the memory scopes of properties. Table II shows it.

The followings depict axioms of the addressing operator &.

$$\&v_1 \neq \&v_2 \tag{1}$$

$$0 \leq e_2 < c \Rightarrow \&v \neq \&e_1[e_2] \tag{2}$$

$$\&e_1[e_2] = \&e_3[e_4] \Leftrightarrow (\&e_1 = \&e_3 \wedge e_2 = e_4 \wedge 0 \leq e_2, e_4 < c) \tag{3}$$

where c is length of the arrays e_1 and e_3.

Axiom (1) states that different variables v_1 and v_2 have different addresses. Axiom (2) states that, if an array subscript is valid, the addresses of a variable v and an array expression $e_1[e_2]$ are different. Axiom (3) states that different components of an array has different addresses. If we are not concerned with array bound checking, we can relax restrictions of array subscripts in Axiom (2) and (3).

For an assignment $e_1 := e_2$, let i, j respectively be the program points before/after this statement. If a property holds at the point i, and $\&e_1$ is not in the memory scope of this property, the property still holds at the point j.

D. Iterative Data-Flow Analysis Framework

Iterative data-flow analysis framework [18] can be used to solve data-flow problems. An iterative data-flow analysis framework can be characterized as a four-tuple $(D, L_{\mathcal{G}}, \sqcap_{\mathcal{G}}, F_{\mathcal{G}})$, where $\mathcal{G}$ represents a control flow graph (CFG), and:

- D is the direction of analysis, which is forward or backward.
- $L_{\mathcal{G}}$ is a description of a meet semi-lattice that represents the data flow values relevant to the problem. The height of $L_{\mathcal{G}}$ must be finite.
- $\sqcap_{\mathcal{G}}$ is a description of the meet operator of the semi-lattice. $\sqcap_{\mathcal{G}}$ is derivable from $L_{\mathcal{G}}$.
- $F_{\mathcal{G}}$ is a description of the set of admissible flow functions from $L_{\mathcal{G}}$ to $L_{\mathcal{G}}$. $F_{\mathcal{G}}$ must be monotonic.

For a data-flow analysis, as long as the four elements are given, the analysis can be performed by an iterative algorithm. If the flow function of a dataflow analysis is monotone, and the height of the lattice is finite, then the analysis will terminate [18].

Our method uses three *forward* data-flow analyses. The *forward* data flow equations are the following forms.

$$In_n = \begin{cases} BI & n \text{ is Start statement} \\ \bigsqcap_{p \in pred(n)} F_p(In_p) & otherwise \end{cases}$$

where n and p are statements, In_n are data flow values before the node n. For a statement n, let $F_n : L_{\mathcal{G}} \to L_{\mathcal{G}}$ denote the flow function that applies the statement. BI is the data flow values before the *Start* statement. $F_{\mathcal{G}}$ consists of flow functions F_p. $\sqcap$ is the meet operator.

III. INDEXES ANALYSIS

Our method is built on "simple programs". For a program or a code snippet, we need to determine which variable is the index, what is initial value of the index, what is step of the index. If there are several indexes in a simple program, we need to get the relations of indexes. Therefore, before array properties analysis, we design two auxiliary indexes analyses to get indexes properties as described above.

A. Determine Indexes

Definition III.1. *A scalar variable* x *is an atomic scalar at a program point* u *if for every sequence* S *of assignment of* x *for every execution path reaching* u, *the first assignment of* S *is to assign a constant to* x, *the residual assignments of* S *are to increase (or decrease)* x *by a fixed constant.*

If a variable is an atomic scalar and it occurs in an array subscript, it is the index. We design an iterative data-flow analysis to discover scalar variables. An atomic scalar is a triple, denoted by *(variable, initial value, fixed constant step)*. In this analysis, $L_{\mathcal{G}}$ is $2^{(\mathbb{S}var, \mathbb{E}xpr, \mathbb{C}on)}$, $\mathbb{S}$var is the set of all scalar variables occurring in $\mathcal{G}$, $\mathbb{E}$xpr is the set of all expressions occurring in $\mathcal{G}$, $\mathbb{C}$on is the set of all integer constants occurring in $\mathcal{G}$. BI is $\emptyset$ (Let's recall that BI is the data flow values before the Start statement).

$F_{\mathcal{G}}$ consists of flow functions $\{F_n \mid n \text{ is a statement}\}$, The definition of F_n is:

$$F_n(In_n) = (In_n - Kill_n(In_n)) \cup Gen_n(In_n)$$

$$Gen_n(\mathbf{x}) = \begin{cases} \{(x, init, \top)\} & n \text{ is } x = init \\ \{(x, init, c)\} & n \text{ is } x = x + c, and\ (x, init, \top\ or\ c) \in \mathbf{x} \\ \emptyset & otherwise \end{cases}$$

$$Kill_n(\mathbf{x}) = \begin{cases} \{(x, init, *)\} & n \text{ is } v = e, and\ v \in \{x\} + Opd(init) \\ \emptyset & otherwise \end{cases}$$

where *Opd(init)* is the operands set of *init*. * denotes any scalar variables.

For any L_1, $L_2 \in L_{\mathcal{G}}$, $L_1 \sqcap_{\mathcal{G}} L_2$ is the set of applying $\overline{\sqcap}$ on every pair of elements respectively from L_1 and L_2. For any atomic scalars $a_1 \in L_1, a_2 \in L_2$, if $a_1 \overline{\sqcap} a_2$ is not $\perp$, $a_1 \overline{\sqcap} a_2 \in L_1 \sqcap_{\mathcal{G}} L_2$.

For any atomic scalar $(x_1, init_1, c_1)$,$(x_2, init_2, c_2)$,

$$(x_1, init_1, c_1) \overline{\sqcap} (x_2, init_2, c_2) = (x_1 \tilde{\sqcap} x_2, init_1 \tilde{\sqcap} init_2, c_1 \tilde{\sqcap} c_2)$$

For any variables or expressions x and y,

$$x \tilde{\sqcap} y = \begin{cases} x & x = y \\ \perp & x \neq y \end{cases}$$

$$x \tilde{\sqcap} \top = x \qquad x \tilde{\sqcap} \perp = \perp$$

If there exists any $\perp$ in a triple, the triple is $\perp$.

The analysis is simple, the convergence and soundness are easy to see.

B. Relation of Indexes

There may be several indexes (or atomic scalars) in a simple program. If two indexes occur in a same loop, there may be an equation about them.

For atomic scalars x_1 and x_2, if for every execution path reaching u, the difference of occurring times between statement $x_1 = x_1 + c_1$ and statement $x_2 = x_2 + c_2$ is a constant d. At u, we have:

$$(x_1 - init_{x_1})/c_1 - (x_2 - init_{x_2})/c_2 = d$$

We design an iterative data-flow analysis to get the difference. We use a triple *(atomic scalar$_1$, atomic scalar$_2$, difference)* to denote the information. In this analysis, $L_{\mathcal{G}}$ is $2^{(\mathbb{A}var, \mathbb{A}var, \mathbb{Z})}$, $\mathbb{A}$var is the set of all atomic scalar variables occurring in $\mathcal{G}$. Before the *Start* statement, all differences are 0. *BI* is $\{(x_1, x_2, 0) \mid x_1, x_2 \in \mathbb{A}var \wedge x_1 \neq x_2\}$. $\sqcap_{\mathcal{G}}$ is $\cap$.

$F_{\mathcal{G}}$ consists of flow functions F_n, which are:

$$F_n(In_n) = (In_n - Kill_n(In_n)) \cup Gen_n(In_n)$$

$$Gen_n(\mathbf{x}) = \begin{cases} \{(*, x, d-1)\} & n \text{ is } x = x + c, and\ (*, x, d) \in \mathbf{x} \\ \{(x, *, d+1)\} & n \text{ is } x = x + c, and\ (x, *, d) \in \mathbf{x} \\ \emptyset & otherwise \end{cases}$$

$$Kill_n(\mathbf{x}) = \begin{cases} \{(*, x, d)\} & n \text{ is } x = x + c, and\ (*, x, d) \in \mathbf{x} \\ \{(x, *, d)\} & n \text{ is } x = x + c, and\ (x, *, d) \in \mathbf{x} \\ \emptyset & otherwise \end{cases}$$

where * denotes any scalar variables.

The analysis is simple, the convergence and soundness are easy to see.

IV. Array Properties Analysis

In this section, we design an iterative data-flow analysis to get properties.

A. Data Flow Values

In the sequel, we assume all expressions are free of side-effects. Expressions with side-effects (e.g. "i++") converted into equivalent expressions without side-effects. Let $\mathbb{E}$xpr be the set of all expressions (all variables are also viewed as expressions) occurring in a CFG $\mathcal{G}$.

Before we give data flow values $L_{\mathcal{G}}$, we need to give two auxiliary definitions: $\mathbb{P}$roperP and $\preccurlyeq$.

$\mathbb{P}$roperP is a set of all properties such that for any $p \in \mathbb{P}$roperP, p satisfies the following conditions:

- if $e_1 \in \mathbb{E}$xpr and $e_2 \in \mathbb{E}$xpr, $(e_1\ op\ e_2) \in \mathbb{P}$roperP.
- if $e_1 \in \mathbb{E}$xpr and $e_2 \in \mathbb{E}$xpr, c is a constant in $\mathcal{G}$, p_1 is in $\mathbb{P}$roperP and different from Φ, $p1[e2/x]$ is also in $\mathbb{P}$roperP, and the depth of nesting of quantifiers inside p_1 is less than the number of *indexes* in $\mathcal{G}$, then $\forall x.x \in [e_1, c, e_2), p_1 \in \mathbb{P}$roperP.
- if $e_1 \in \mathbb{E}$xpr, $e_2 \in \mathbb{E}$xpr, and c is a constant in $\mathcal{G}$, $(\forall x.x \in [e_1, c, e_2), \phi) \in \mathbb{P}$roperP.
- if $p_1 \| p_2$ is a condition expression occurring in $\mathcal{G}$, $p_1 \vee p_2 \in \mathbb{P}$roperP.

Corollary IV.1. *Let $\mathcal{G}$ be a CFG. The size of $\mathbb{P}$roperP is finite.*

Corollary IV.1 follows directly from the definition of $\mathbb{P}$roperP and $\mathbb{E}$xpr.

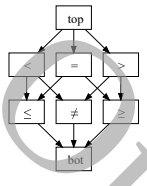

Figure 1. Lattice on Op, $(Op, \sqsubseteq_{op}, \sqcap_{op}, top, bot)$

Now, we define $\preccurlyeq$ relations. For any properties p_1, p_2, we say

$$p_2 \preccurlyeq p_1$$

when the following criteria are satisfied:

- if p_2 is a *simple property* $(e_3\ op_2\ e_4)$, p_1 is a *simple property* $(e_1\ op_1\ e_2)$ such that:
  - $e_1 = e_3 \wedge e_2 = e_4 \wedge op_2 \sqsubseteq_{op} op_1$, or
  - $e_1 = e_4 \wedge e_2 = e_3 \wedge \backsim op_2 \sqsubseteq_{op} op_1$

  where $\sqsubseteq_{op}$ is defined in Fig. 1. $\backsim op$ is defined as:

$$\begin{cases} \backsim < = >, \backsim \leq = \geq, \backsim > = <, \backsim \geq = \leq \\ \backsim op = op \quad otherwise \end{cases}$$

- if p_2 is $\forall k.k \in [e_3, c_2, e_4), \phi$, $p_1 = p_2$.

- if p_2 is $\forall k.k \in [e_3, c_2, e_4), p_4$ such that $p_4 \neq \phi$, p_1 is
  - $\forall k.k \in [e_1, c_1, e_2), p_3$ such that (a) $p_3 \neq \phi$, (b) $[e_1, c_1, e_2)$=$[e_3, c_2, e_4)$, and (c) $p_4 \preccurlyeq p_3$, or
  - $\forall k.k \in [e_1, c_1, e_2), \phi$ such that $[e_1, c_1, e_2)$=$[e_3, c_2, e_4)$.
- if p_2 is a disjunction property, p_1 is a disjunction property such that $p_1 = p_2$.

Intuitively, if p_2 is not $\forall k.k \in [e_1, c_1, e_2), \phi$, $p_1 \preccurlyeq p_2$ means $p_2 \Rightarrow p_1$.

Corollary IV.2. *For any properties p_1, p_2 and p_3,*

1) $p_1 \preccurlyeq p_2 \wedge p_2 \preccurlyeq p_1 \Rightarrow p_1 = p_2$
2) $p_1 \preccurlyeq p_2 \wedge p_2 \preccurlyeq p_3 \Rightarrow p_1 \preccurlyeq p_3$
3) $p_1 \preccurlyeq p_2 \Rightarrow \mathbb{M}(p_1) \subseteq \mathbb{M}(p_2)$.*(Note that, $\mathbb{M}(\phi)$ is $\emptyset$.)*

Now, we can give the $L_{\mathcal{G}}$. In this analysis, $L_{\mathcal{G}}$ is

$$\{S \mid S \subseteq \mathbb{P}\text{roperP} \wedge \forall p_1, p_2 \in S, p_1 \preccurlyeq p_2 \Rightarrow p_1 = p_2\}$$

For any $L_1 \in L_{\mathcal{G}}$, for any properties $p_1, p_2 \in L_1$ such that $p_1 \neq p_2$, there is not the $\preccurlyeq$ relation between p_1 and p_2.

The partial order $\sqsubseteq$ is defined as follows. For any $L_1, L_2 \in L_{\mathcal{G}}$, we say

$$L_1 \sqsubseteq L_2$$

if for each property $p_1 \in L_1$, there is a property $p_2 \in L_2$ such that $p_1 \preccurlyeq p_2$.

For any $L \in L_{\mathcal{G}}$, $\emptyset \sqsubseteq L$. *BI* is $\emptyset$. (Let's recall that *BI* the data flow values before the Start statement).

B. Meet Operations

Before giving meet operations $\sqcap_{\mathcal{G}}$, we first define the $\overline{\sqcap}$ operations over the properties in $\mathbb{P}$roperP. $\overline{\sqcap}$ is defined as follows:

- $(e_1\ op_1\ e_2)\ \overline{\sqcap}\ (e_3\ op_2\ e_4)$.
  - if $e_1 = e_3 \wedge e_2 = e_4$, and $op_1 \sqcap_{op} op_2$ is not *bot*, the meet result is $(e_1\ op_r\ e_2)$, where $op_r = op_1 \sqcap_{op} op_2$.
  - if $e_1 = e_4 \wedge e_2 = e_3$, and $op_1 \sqcap_{op} \backsim op_2$ is not *bot*, the meet result is $(e_1\ op_r\ e_2)$, where $op_r = op_1 \sqcap_{op} \backsim op_2$.
  - otherwise, the meet result is $\perp$.

  where $\sqcap_{op}$ is defined in Fig. 1. $\backsim op$ is defined in IV-A.
- $\forall x.x \in [e_1, c_1, e_2), p_1 \overline{\sqcap} \forall x.x \in [e_3, c_2, e_4), p_2$.

  $$\forall x.x \in [e_1, c_1, e_2), p_1 \overline{\sqcap} \forall x.x \in [e_3, c_2, e_4), p_2 = \begin{cases} \forall x.x \in [e_1, c_1, e_2), p_1 \overline{\sqcap} p_2 & [e_1, c_1, e_2) = [e_3, c_2, e_4) \\ \perp & otherwise \end{cases}$$

  Note that, if $p_1 \overline{\sqcap} p_2 = \perp$, the meet result is $\perp$.
- otherwise, if $p_1 \preccurlyeq p_2$, $p_1 \overline{\sqcap} p_2$ is p_1; if $p_2 \preccurlyeq p_1$, $p_1 \overline{\sqcap} p_2$ is p_2; otherwise, $p_1 \overline{\sqcap} p_2$ is $\perp$.

Intuitively, if p_1 and p_2 are not $\forall k.k \in [e_1, c_1, e_2), \phi$, and $p_1 \overline{\sqcap} p_2$ is not $\perp$, $p_1 \overline{\sqcap} p_2$ is equivalent to $p_2 \vee p_1$.

Corollary IV.3. *For any properties p_1, p_2 and p_3,*

1) *if $p_1 \overline{\sqcap} p_2$ is not $\perp$,*

$$p_1 \overline{\sqcap} p_2 \preccurlyeq p_1 \wedge p_1 \overline{\sqcap} p_2 \preccurlyeq p_2$$
$$p_1 \vee p_2 \Rightarrow p_1 \overline{\sqcap} p_2$$

2) $p_1 \preccurlyeq p_2 \Leftrightarrow p_1 = p_1 \overline{\sqcap} p_2$
3) $p_1 \preccurlyeq p_2 \wedge p_1 \preccurlyeq p_3 \Rightarrow p_1 \preccurlyeq p_2 \overline{\sqcap} p_3$

Corollary IV.4. *For any properties p_1, p_2, p_3, $\overline{\sqcap}$ holds the following property:*

1) $p_1 \overline{\sqcap} p_1 = p_1$
2) $p_1 \overline{\sqcap} p_2 = p_2 \overline{\sqcap} p_1$
3) $(p_1 \overline{\sqcap} p_2) \overline{\sqcap} p_3 = p_1 \overline{\sqcap} (p_2 \overline{\sqcap} p_3)$

Now, we define $\sqcap_{\mathcal{G}}$.

$$L_1 \sqcap_{\mathcal{G}} L_2 \stackrel{\text{def}}{=} Reduce(\{p_1 \overline{\sqcap} p_2 \mid p_1 \in L_1 \wedge p_2 \in L_2 \wedge p_1 \overline{\sqcap} p_2 \neq \perp\})$$

For any $S \in 2^{\mathbb{P}roperP}$, the *Reduce(S)* function is defined as:

$$Reduce(S) = S \backslash \{p \mid p \in S \wedge \exists p_1 \in S, p_1 \neq p \Rightarrow p \preccurlyeq p_1\}$$

Intuitively, for any $S \in 2^{\mathbb{P}roperP}$, the *Reduce* function makes $Reduce(S) \in L_{\mathcal{G}}$ hold.

Corollary IV.5. *For any S_1, $S_2 \in 2^{\mathbb{P}roperP}$,*

1) $Reduce(S_1) \in L_{\mathcal{G}}$
2) $S_1 \sqsubseteq Reduce(S_1) \wedge Reduce(S_1) \sqsubseteq S_1$
3) $S_1 \sqsubseteq S_2 \Rightarrow Reduce(S_1) \sqsubseteq Reduce(S_2)$

Lemma IV.1. *For any L_1, $L_2 \in L_{\mathcal{G}}$, $L_1 \sqcap_{\mathcal{G}} L_2 \in L_{\mathcal{G}}$.*

Proof. The conclusions follow directly from the definition of $\sqcap_{\mathcal{G}}$. □

Lemma IV.2. *For any $S_1, S_2, S_3 \in 2^{\mathbb{P}roperP}$,*

1) $S_1 \sqcap_{\mathcal{G}} S_2 \sqsubseteq S_1 \wedge S_1 \sqcap_{\mathcal{G}} S_2 \sqsubseteq S_2$
2) $S_1 \sqsubseteq S_2 \wedge S_1 \sqsubseteq S_3 \Rightarrow S_1 \sqsubseteq S_2 \sqcap_{\mathcal{G}} S_3$

Proof. According to Corollary IV.3, for any $p_1 \in S_1, p_2 \in S_2$, if $p_1 \overline{\sqcap} p_2 \in S_1 \sqcap_{\mathcal{G}} S_2$, then $p_1 \overline{\sqcap} p_2 \preccurlyeq p_1 \wedge p_1 \overline{\sqcap} p_2 \preccurlyeq p_2$. We have: $S_1 \sqcap_{\mathcal{G}} S_2 \sqsubseteq S_1 \wedge S_1 \sqcap_{\mathcal{G}} S_2 \sqsubseteq S_2$. Thus, the conclusion (1) holds.

$S_1 \sqsubseteq S_2 \wedge S_1 \sqsubseteq S_3$ implies $\forall p_1 \in S_1, \exists p_2 \in S_2, \exists p_3 \in S_3 \Rightarrow p_1 \preccurlyeq p_2 \wedge p_1 \preccurlyeq p_3$. According to Corollary IV.3, $p_1 \preccurlyeq p_2 \overline{\sqcap} p_3$. We have: $S_1 \sqsubseteq S_2 \sqcap_{\mathcal{G}} S_3$. Thus, the conclusion (2) holds. □

Lemma IV.3. *For any L_1, $L_2 \in L_{\mathcal{G}}$, $L_1 \sqsubseteq L_2 \wedge L_2 \sqsubseteq L_1 \Leftrightarrow L_1 = L_2$.*

Proof. $L_1 \sqsubseteq L_2 \wedge L_2 \sqsubseteq L_1$ implies $\forall p_1 \in L_1, \exists p_2 \in L_2, \exists p_3 \in L_1 \Rightarrow p_1 \preccurlyeq p_2 \wedge p_2 \preccurlyeq p_3$. $p_1 \preccurlyeq p_2 \wedge p_2 \preccurlyeq p_3 \wedge L_1 \in L_{\mathcal{G}} \Rightarrow p_1 = p_3 \Rightarrow p_1 \preccurlyeq p_2 \wedge p_2 \preccurlyeq p_1 \Rightarrow p_1 = p_2$. Therefore, $\forall p_1 \in L_1, \exists p_2 \in L_2 \Rightarrow p_1 = p_2 \Rightarrow p_1 \in L_2$.

$L_1 \sqsubseteq L_2 \wedge L_2 \sqsubseteq L_1$ also implies $\forall p_2 \in L_2, \exists p_1 \in L_1, \exists p_3 \in L_2 \Rightarrow p_2 \preccurlyeq p_1 \wedge p_1 \preccurlyeq p_3$. $p_2 \preccurlyeq p_1 \wedge p_1 \preccurlyeq p_3 \wedge L_2 \in L_{\mathcal{G}} \Rightarrow p_2 = p_3 \Rightarrow p_2 \preccurlyeq p_1 \wedge p_1 \preccurlyeq p_2 \Rightarrow p_1 = p_2$. Therefore, $\forall p_2 \in L_2, \exists p_1 \in L_1 \Rightarrow p_1 = p_2 \Rightarrow p_2 \in L_1$. Thus, $L_1 = L_2$. □

Lemma IV.4. *For any L_1, $L_2 \in L_{\mathcal{G}}$, $L_1 \sqsubseteq L_2 \Leftrightarrow L_1 = L_1 \sqcap_{\mathcal{G}} L_2$.*

Proof. $L_1 \sqsubseteq L_2 \Leftrightarrow L_1 = L_1 \sqcap_{\mathcal{G}} L_2$ iff

1) $L_1 = L_1 \sqcap_{\mathcal{G}} L_2 \Rightarrow L_1 \sqsubseteq L_2$
2) $L_1 \sqsubseteq L_2 \Rightarrow L_1 = L_1 \sqcap_{\mathcal{G}} L_2$

The conclusion (1) follows directly from Lemma IV.2.

Let $S(L_1, L_2)$ be $\{p_1 \overline{\sqcap} p_2 \mid p_1 \in L_1 \wedge p_2 \in L_2 \wedge p_1 \overline{\sqcap} p_2 \neq \bot\}$.

According to Corollary IV.5 and the definition of $S(L_1, L_2)$, $L_1 \sqsubseteq L_2 \Rightarrow L_1 \sqsubseteq S(L_1, L_2) \Rightarrow L_1 \sqsubseteq Reduce(S(L_1, L_2)) \Rightarrow L_1 \sqsubseteq L_1 \sqcap_{\mathcal{G}} L_2$.

According to Lemma IV.2, $L_1 \sqcap_{\mathcal{G}} L_2 \sqsubseteq L_1$.

According to Lemma IV.3, $L_1 \sqsubseteq L_1 \sqcap_{\mathcal{G}} L_2 \wedge L_1 \sqcap_{\mathcal{G}} L_2 \sqsubseteq L_1 \wedge L_1 \in L_{\mathcal{G}} \wedge L_1 \sqcap_{\mathcal{G}} L_2 \in L_{\mathcal{G}}$ implies $L_1 = L_1 \sqcap_{\mathcal{G}} L_2$. The conclusion (2) holds. □

Theorem IV.1. *($L_{\mathcal{G}}$,$\sqcap_{\mathcal{G}}$) is a meet semi-lattice on a CFG $\mathcal{G}$.*

Proof. ($L_{\mathcal{G}}$,$\sqcap_{\mathcal{G}}$) is a meet semi-lattice iff, for all $L_1, L_2, L_3 \in L_{\mathcal{G}}$,

1) $L_1 \sqcap_{\mathcal{G}} L_1 = L_1$
2) $L_1 \sqcap_{\mathcal{G}} L_2 = L_2 \sqcap_{\mathcal{G}} L_1$
3) $(L_1 \sqcap_{\mathcal{G}} L_2) \sqcap_{\mathcal{G}} L_3 = L_1 \sqcap_{\mathcal{G}} (L_2 \sqcap_{\mathcal{G}} L_3)$
4) $L_1 \sqsubseteq L_2 \Leftrightarrow L_1 = L_1 \sqcap_{\mathcal{G}} L_2$

The conclusions (1) and (2) follow directly from Lemma IV.4 and the definition of $\sqcap_{\mathcal{G}}$. The conclusion (4) has been proved in Lemma IV.4.

Let $S(L_1, L_2)$ be $\{p_1 \overline{\sqcap} p_2 \mid p_1 \in L_1 \wedge p_2 \in L_2 \wedge p_1 \overline{\sqcap} p_2 \neq \bot\}$. To prove conclusion (3), we just need to prove that:

$$(L_1 \sqcap_{\mathcal{G}} L_2) \sqcap_{\mathcal{G}} L_3 = Reduce(S(S(L_1, L_2), L_3))$$
$$Reduce(S(S(L_1, L_2), L_3)) = Reduce(S(L_1, S(L_2, L_3)))$$
$$Reduce(S(L_1, S(L_2, L_3))) = L_1 \sqcap_{\mathcal{G}} (L_2 \sqcap_{\mathcal{G}} L_3)$$

$(L_1 \sqcap_{\mathcal{G}} L_2) \sqcap_{\mathcal{G}} L_3 \sqsubseteq L_1 \sqcap_{\mathcal{G}} L_2 \sqsubseteq S(L_1, L_2)$.
$(L_1 \sqcap_{\mathcal{G}} L_2) \sqcap_{\mathcal{G}} L_3 \sqsubseteq L_3$.
According to Lemma IV.2, $(L_1 \sqcap_{\mathcal{G}} L_2) \sqcap_{\mathcal{G}} L_3 \sqsubseteq S(L_1, L_2) \sqcap_{\mathcal{G}} L_3$.
$S(L_1, L_2) \sqcap_{\mathcal{G}} L_3 \sqsubseteq S(L_1, L_2) \sqsubseteq L_1 \sqcap_{\mathcal{G}} L_2$.
$S(L_1, L_2) \sqcap_{\mathcal{G}} L_3 \sqsubseteq L_3$.
According to Lemma IV.2, $S(L_1, L_2) \sqcap_{\mathcal{G}} L_3 \sqsubseteq (L_1 \sqcap_{\mathcal{G}} L_2) \sqcap_{\mathcal{G}} L_3$.
Thus, $(L_1 \sqcap_{\mathcal{G}} L_2) \sqcap_{\mathcal{G}} L_3 = S(L_1, L_2) \sqcap_{\mathcal{G}} L_3 = Reduce\ (S(S(L_1, L_2), L_3))$.
For $Reduce(S(L_1, S(L_2, L_3))) = L_1 \sqcap_{\mathcal{G}} (L_2 \sqcap_{\mathcal{G}} L_3)$, the proof is similar to $(L_1 \sqcap_{\mathcal{G}} L_2) \sqcap_{\mathcal{G}} L_3 = Reduce(S(S(L_1, L_2), L_3))$.
According to the definition of S, $S(S(L_1, L_2), L_3) = S(L_1, S(L_2, L_3))$ holds.
Thus, the conclusion (3) holds.

□

Theorem IV.2. *Let $\mathcal{G}$ be a CFG. The height of meet semi-lattice ($L_{\mathcal{G}}$,$\sqcap_{\mathcal{G}}$) is finite.*

Proof. $L_{\mathcal{G}}$ is

$$\{S \mid S \subseteq \mathbb{P}\text{roperP} \wedge \forall p_1, p_2 \in S, p_1 \preccurlyeq p_2 \Rightarrow p_1 = p_2\}$$

We have: $L_{\mathcal{G}} \subseteq 2^{\mathbb{P}\text{roperP}}$. The size of $\mathbb{P}$roperP is finite according to Corollary IV.1, thus the height of meet semi-lattice ($L_{\mathcal{G}}$,$\sqcap_{\mathcal{G}}$) is finite. □

C. Flow Function

Let a be data flow values before a statement n. The flow function $F_n(a)$ is defined as:

$$F_n(a) = \begin{cases} Reduce(Transfer(n, a, Produce(n, a))) & if\ n\ is\ e_1[e_2] := e_3 \\ Reduce(GenPhi(n, Produce(n, a))) & if\ n\ is\ index\ initialization \\ Reduce(HandlePhi(n, a, Produce(n, a))) & if\ n\ is\ index\ update\ stmt \\ Reduce(Produce(n, a)) & otherwise \end{cases}$$

$$Produce(n, a) = Propagated(a \cup Semantics(n, a))$$

For any statement n, F_n first computes the semantics of n and produces as many as possible properties according to propagated rules (*Produce* function). If n is $e_1[e_2] := e_3$, *Produce(n,a)* may lose universally quantified properties, thus we need to transfer them to $F_n(a)$ (*Transfer* function). If n is an index initialization statement, $F_n(a)$ produces ϕ properties (*GenPhi* function). If n is an index update statement, $F_n(a)$ produces universally quantified properties by handling ϕ properties (*HandlePhi* function). The *Reduce* function is defined in section IV-B. Intuitively, for any $S \in 2^{\mathbb{P}roperP}$, *Reduce* function makes $Reduce(S) \in L_{\mathcal{G}}$ hold. The detailed descriptions of these functions are given as follows.

1) Semantics Function: Let a be data flow values before a statement n. *Semantics(n,a)* function computes the relations between input and output states.

- If $n \stackrel{\text{def}}{=} lh := e$, $Semantics(n, a)$ is defined as:

  $$\{lh' = e\} \cup \{lh'_i = lh_i \mid \&lh \neq \&lh_i \wedge lh_i \in \mathbb{L}\text{H}\}$$

  where lh_i represents a left-hand expression, $\mathbb{L}$H is the set of all left-hand expressions occurring in a program.
  We use unprimed variables to denote pre-state quantities, and primed variables to denote post-state quantities.
- Let *cond* be the condition expression of "while" and "if" statements. If n is *cond*,
  - in the *true* branch, $Semantics(n, a) = \{cond\} \cup \{lh'_i = lh_i \mid lh_i \in \mathbb{L}\text{H}\}$.
  - in the *false* branch, $Semantics(n, a) = \{\neg cond\} \cup \{lh'_i = lh_i \mid lh_i \in \mathbb{L}\text{H}\}$.

Lemma IV.5. *For any $L_1, L_2 \in L_{\mathcal{G}}$, if $L_1 \sqsubseteq L_2$, $Semantics(n, L_1) \sqsubseteq Semantics(n, L_2)$.*

Proof. According to the definition of *Semantics* function,

- if n is *cond*, the conclusion follows directly from the definition of *Semantics* function.
- if n is $lh := e$, for any $lh_i \in \mathbb{L}\text{H}$, if $\bigwedge L_1 \Rightarrow \&lh \neq \&lh_i$, then $\bigwedge L_2 \Rightarrow \&lh \neq \&lh_i$. Thus, $Semantics(n, L_1) \sqsubseteq Semantics(n, L_2)$.

Thus, $Semantics(n, L_1) \sqsubseteq Semantics(n, L_2)$. □

After *Semantics(n,a)*, we transfer the result of *Semantics(n,a)* and a to *propagated* function to produce as many as possible properties.

2) Propagated Function: Propagated function is defined as:

$$Propagated(a \cup Semantics(n,a)) = Filter(ApplyRules(a \cup Semantics(n,a)))$$

Propagated function does two things:

1) producing as many as possible properties according to propagated rules (*ApplyRules* function).
2) filtering out pre-state properties. The inputs of *Propagated* function are the result of *Semantics(n,a)* and a. There are pre-state quantities in the result of *Semantics(n,a)*. Thus, after (1), we need to filter out pre-state properties and fetch post-state properties (*Filter* function).

Fig. 2 shows some of propagated rules. Note that all rules satisfy **propagated rules constraints** described as following.

Notice that, $>, \geq$ and $<, \leq$ are respectively dual, so we just describe $<, \leq$ here. Let *in* be the input properties set of a rule, and *out* be the output properties set of the rule. Generally, a rule can be defined as:

$$\bigwedge in \Rightarrow \bigwedge out$$

1) If $in \subseteq \mathbb{P}$roperP, $out \subseteq \mathbb{P}$roperP.
2) If $p \in in$ such that p is $e_1 \leq e_2$, two additional rules $\bigwedge in[e_1 < e_2/e_1 \leq e_2] \Rightarrow \bigwedge out_1$ and $\bigwedge in[e_1 = e_2/e_1 \leq e_2] \Rightarrow \bigwedge out_2$ should be provided, where $out \sqsubseteq out_1$ and $out \sqsubseteq out_2$.
   For example, $e_1 \leq e_2$ is in **LE-1**, thus, two additional rules **LT-1** and **ASSIGN-2** are provided. We refer to **LT-1** and **ASSIGN-2** as the corresponding rules of **LE-1**.
3) If $p \in in$ such that p is $e_1 \neq e_2$, another rule $\bigwedge in[e_1 < e_2/e_1 \neq e_2] \Rightarrow \bigwedge out_1$ should be provided, where $out \sqsubseteq out_1$.
4) If $p \in in$ such that p is $\forall x.x \in [e_1, c, e_2), p_1$, another rule $\bigwedge in[\forall x.x \in [e_1, c, e_2), \phi / \forall x.x \in [e_1, c, e_2), p_1] \Rightarrow \bigwedge out_1$ should be provided, where $out \sqsubseteq out_1$. For example, **LT-4** and **LT-5** are respectively the corresponding rules of **LT-2** and **LT-3**.

Lemma IV.6. *For any properties sets S_1 and S_2, if $S_1 \sqsubseteq S_2$, $Propagated(S_1) \sqsubseteq Propagated(S_2)$.*

Proof. For any $in_1 \subseteq S_1$ if a rule r is applied to in_1, and the application produces an output properties set out_1, according to the definition of $\preccurlyeq$ and **propagated rules constraints** (2), (3) and (4), there is a set $in_2 \subseteq S_2$ and a corresponding rule r' such that (a) $in_1 \sqsubseteq in_2$, and (b) r' can be applied to in_2, and (c) the application produces an output properties set out_2 and $out_1 \sqsubseteq out_2$. Thus, for any properties set S_1 and S_2, if $S_1 \sqsubseteq S_2$, $Propagated(S_1) \sqsubseteq Propagated(S_2)$. □

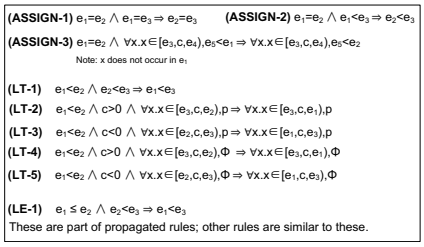

Figure 2. Propagated Rules

3) Transfer Function: If n is $e_1[e_2] := e_3$, *Produce* function can lose any universally quantified property p which contains e_1, even if $\&e_1[e_2] \notin \mathbb{M}(p)$. Thus, we must transfer p to $F_n(a)$ if $\&e_1[e_2] \notin \mathbb{M}(p)$. For any $S \in 2^{\mathbb{P}roperP}$, $Transfer(e_1[e_2] := e_3, a, S)$ is defined as:

$$Transfer(e_1[e_2] := e_3, a, S) = S \cup \{p \mid p \in a \wedge p \text{ is a } UQProp \wedge \&e_1[e_2] \notin \mathbb{M}(p)\}$$

where *UQProp* is the abbreviation of "universally quantified property".

Lemma IV.7. *For any $x, y \in L_{\mathcal{G}}$, $S_1, S_2 \in 2^{\mathbb{P}roperP}$, if $x \sqsubseteq y$ and $S_1 \sqsubseteq S_2$, $Transfer(e_1[e_2] := e_3, x, S_1) \sqsubseteq Transfer(e_1[e_2] := e_3, y, S_2)$.*

Proof. Let $TS(x)$ be $\{p \mid p \in x \wedge p \text{ is a } UQProp \wedge \&e_1[e_2] \notin \mathbb{M}(p)\}$. According to definition of *Transfer*, we just need to prove $TS(x) \sqsubseteq TS(y)$. For any property $p_1 \in TS(x)$, there is a universally quantified property $p_2 \in y$ such that $p_1 \preccurlyeq p_2$. According to Corollary IV.2, we have: $\mathbb{M}(p_1) \subseteq \mathbb{M}(p_2)$. $\bigwedge x \Rightarrow \&e_1[e_2] \notin \mathbb{M}(p_1)$ implies $\bigwedge y \Rightarrow \&e_1[e_2] \notin \mathbb{M}(p_2)$, which implies $p_2 \in TS(y)$. Therefore, $TS(x) \sqsubseteq TS(y)$. □

Example IV.1. *Consider a statement $n \stackrel{def}{=} A[i] := x$, we would like to compute $F_n(a)$, where $a = \{\forall k.k \in [0, 1, j), A[k] = x \wedge i \geq j\}$.*
$ApplyRules(a \cup Semantics(n,a)) = \{(A[i])' = x', (\forall k.k \in [0, 1, j'), A[k] = x'), i' \geq j'\} \cup Semantics(n,a)$.
$Produce(n,a) = \{A[i] = x, i \geq j\}$.
The $Produce(n,a)$ loses the universally quantified property since $(\forall k.k \in [0, 1, j'), A[k] = x')$ is not the post-state (the array A is not post-state quantity). However, $\&A[i] \notin \mathbb{M}(\forall k.k \in [0, 1, j), A[k] = x)$. So we transfer it to $F_n(a)$. $F_n(a) = \{A[i] = x, i \geq j, (\forall k.k \in [0, 1, j), A[k] = x)\}$.

4) GenPhi Function: For index initialization $n \stackrel{\text{def}}{=} i := init$, $S \in 2^{\mathbb{P}roperP}$, *GenPhi* is defined as:

$$GenPhi(n, S) = \{\forall k.k \in [init, c, i), \phi\} \cup S$$

We use ϕ to denote a special property. c is the step of the index i (we obtain it by the auxiliary indexes analyses described in Section III). $\forall k.k \in [init, step, i), \phi$ holds since $[init, step, i)$ is $\emptyset$.

Corollary IV.6. *For any $S_1, S_2 \in 2^{\mathbb{P}roperP}$, if n is an index initialization statement and $S_1 \sqsubseteq S_2$, $GenPhi(n, S_1) \sqsubseteq GenPhi(n, S_2)$.*

5) HandlePhi function: For index update statement $n \stackrel{\text{def}}{=} i := i + c$, $S \in 2^{\mathbb{P}roperP}$, *HandlePhi* defined as:

$$HandlePhi(n, a, S) = \begin{cases} S \cup \{\forall x.x \in [init_i, c, i), \psi(\cdots, e_1[f_1(x)], \cdots)\} if\ (4)\ or\ (5)\ holds \\ S \quad otherwise \end{cases}$$

where ψ is a property about array elements. f_1 is a general index function. Notice that $\psi(\cdots, e_1[f_1(x)], \cdots)$ does not contain i.

$$\begin{aligned} i = init_i \in a \wedge \forall x.x \in [init_i, c, i), \phi \in a \\ \wedge \psi(\cdots, e_1[f_1(i)], \cdots) \in a \qquad (4) \\ \forall x.x \in [init_i, c, i), \psi_1(\cdots, e_1[f_1(x)], \cdots) \in a \\ \wedge \psi_2(\cdots, e_1[f_1(i)], \cdots) \in a \\ \wedge \psi(\cdots, e_1[f_1(x)], \cdots) = \\ \psi_1(\cdots, e_1[f_1(x)], \cdots) \overline{\sqcap} \psi_2(\cdots, e_1[f_1(x)], \cdots) \qquad (5) \end{aligned}$$

If condition (4) holds, $[init_i, c, i)$ is $\emptyset$ since $i = init_i$ holds. Thus after n, $\forall x.x \in [init_i, c, i), \psi(\cdots, e_1[f_1(x)], \cdots)$ holds. If condition (5) holds,

$$\psi(\cdots, e_1[f_1(x)], \cdots) = \psi_1(\cdots, e_1[f_1(x)], \cdots) \overline{\sqcap} \psi_2(\cdots, e_1[f_1(x)], \cdots)$$

implies $\forall x.x \in [init_i, c, i), \psi_1(\cdots, e_1[f_1(x)], \cdots) \Rightarrow \forall x.x \in [init_i, c, i), \psi(\cdots, e_1[f_1(x)], \cdots)$ and $\psi_2(\cdots, e_1[f_1(i)], \cdots) \Rightarrow \psi(\cdots, e_1[f_1(i)], \cdots)$. After n, $[init_i, c, i') = [init_i, c, i + c)$, which means i (pre-state quantity) is added to $[init_i, c, i)$. Thus after n, $\forall x.x \in [init_i, c, i), \psi(\cdots, e_1[f_1(x)], \cdots)$ holds.

Lemma IV.8. *For any $x, y \in L_{\mathcal{G}}$, $S_1, S_2 \in 2^{\mathbb{P}roperP}$, if* n *is $i := i + c$ and $x \sqsubseteq y$ and $S_1 \sqsubseteq S_2$, $HandlePhi(n, x, S_1) \sqsubseteq HandlePhi(n, y, S_2)$.*

Proof. Let $\psi(k)$ be abbreviation of $\psi(\cdots, e_1[f_1(k)], \cdots)$.

- if condition (4) holds in x, $HandlePhi(n, x, S_1) = S_1 \cup \{\forall k.k \in [init_i, c, i), \psi(k)\}$. Condition (4) implies $i = init_i \in y \wedge \forall k.k \in [init_i, c, i), \phi \in y \wedge \psi'(i) \in y$ such that $\psi(i) \preccurlyeq \psi'(i)$. We have: $HandlePhi(n, y, S_2) = L_2 \cup \{\forall k.k \in [init_i, c, i), \psi'(k)\}$. Thus, $HandlePhi(n, x, S_1) \sqsubseteq HandlePhi(n, y, S_2)$
- if condition (5) holds in x, $HandlePhi(n, x, S_1) = S_1 \cup \{\forall k.k \in [init_i, c, i), \psi(k)\}$. Condition (5) implies $\forall k.k \in [init_i, c, i), \psi'_1(k) \in a \wedge \psi'_2(i) \in a$ such that $\psi_1(k) \preccurlyeq \psi'_1(k) \wedge \psi_2(i) \preccurlyeq \psi'_2(i)$. $\psi(k) = \psi_1(k) \overline{\sqcap} \psi_2(k) \Rightarrow \psi(k) \preccurlyeq \psi_1(k) \wedge \psi(k) \preccurlyeq \psi_2(k) \Rightarrow \psi(k) \preccurlyeq \psi'_1(k) \wedge \psi(k) \preccurlyeq \psi'_2(k) \Rightarrow \psi(k) \preccurlyeq \psi'_1(k) \overline{\sqcap} \psi'_2(k)$. Let $\psi'(k)$ be $\psi'_1(k) \overline{\sqcap} \psi'_2(k)$. We have: $HandlePhi(n, y, L_2) = S_2 \cup \{\forall k.k \in [init_i, c, i), \psi'(k)\}$. Thus, $HandlePhi(n, x, S_1) \sqsubseteq HandlePhi(n, y, S_2)$

□

Lemma IV.9. *Let* **a** *be data flow values before a statement* **n** *in a CFG $\mathcal{G}$. If $\boldsymbol{a} \in L_{\mathcal{G}}$, $F_n(\boldsymbol{a}) \in L_{\mathcal{G}}$.*

Proof. The conclusion follows directly from the definition of $F_n(\mathbf{a})$. □

Theorem IV.3. *Let* n *be a statement in a CFG $\mathcal{G}$. F_n is monotonic.*

Proof. F_n is monotonic iff $\forall x, y \in L_{\mathcal{G}} : \ x \sqsubseteq y \Rightarrow F_n(x) \sqsubseteq F_n(y)$. According to the definition of F_n, we just need to prove that all functions occurring in F_n are monotonic. We have proved that all functions occurring in F_n are monotonic (Lemma IV.5, Lemma IV.6,Lemma IV.7, Corollary IV.6 , Lemma IV.8, Corollary IV.5). □

D. A Whole Example

Example IV.2. *We give now a whole analysis on "arrayPartCopy" program (depicted in section II). The analysis provides:*

- *At the first iteration:*
  - *after line 2: statement $i := 0$ is an index initialization, so $i = 0, (\forall k.k \in [0, 2, i), \phi)$.*
  - *after line 4: $i = 0$, $(\forall k.k \in [0, 2, i), \phi)$, $i < size$, $A[i] = B[i]$.*
  - *after line 5: statement $i := i + 2$ is an index update statement, $i = 0 \wedge \forall k.k \in [0, 2, i), \phi \wedge A[i] = B[i]$ implies $\forall k.k \in [0, 2, i), A[k] = B[k]$.*
  - *after line 6: $(\forall k.k \in [0, 2, i), A[k] = B[k])$, $i \geq size$, which implies $(\forall k.k \in [0, 2, size), A[k] = B[k])$.*
- *At the second iteration:*
  - *at line 3: $\forall k.k \in [0, 2, i), A[k] = B[k]$, since result of $\forall k.k \in [0, 2, i), \phi \overline{\sqcap} \forall k.k \in [0, 2, i), A[k] = B[k]$ is $\forall k.k \in [0, 2, i), A[k] = B[k]$.*
  - *after line 5: $\forall k.k \in [0, 2, i), A[k] = B[k]$ still holds.*
  - *after line 6: $(\forall k.k \in [0, 2, i), A[k] = B[k])$, $i \geq size$, $(\forall k.k \in [0, 2, size), A[k] = B[k])$, and the iteration converges.*
- *So, the final result at the end of the program is*
  - *$(\forall k.k \in [0, 2, i), A[k] = B[k]) \wedge i \geq size \wedge (\forall k.k \in [0, 2, size), A[k] = B[k])$*

E. Termination

In this section, we shall prove that array properties analysis will terminate. According to iterative data-flow analysis framework [15], we need to prove that the flow function F_n is monotonic (Theorem IV.3), and the height of meet semi-lattice is finite (Theorem IV.2). Thus, array properties analysis will terminate.

F. Soundness

Some notes: let *States* denote the set of states of a program. A state is a tuple (ρ_v, ρ_a), where ρ_v maps scalar variables names to their values, ρ_a maps array names to their values. The semantics of statements of "simple programs" are described in Fig. 3 as functions from *States* to *States*. We use "$i := e$" to denote an assignment to a scalar variable, and use "$A[e_1] := e_2$" to denote an assignment to an array element. Here, we simplify arrays for one dimension arrays.

$$[\![i := e]\!](\rho_v, \rho_a) = (\rho_v[[\![e]\!](\rho_v, \rho_a)/i], \rho_a)$$
$$[\![A[e_1] := e_2]\!](\rho_v, \rho_a) = (\rho_v, \rho_a[F/A])$$
$$\text{where } F = \lambda z. \begin{cases} \rho_a(A)(z) & if\ z \neq [\![e_1]\!](\rho_v, \rho_a) \\ [\![e_2]\!](\rho_v, \rho_a) & otherwise \end{cases}$$
$$[\![\textbf{while}(cond)\ stmt]\!](\rho_v, \rho_a) =$$
$$\begin{cases} (\rho_v, \rho_a) & if\ [\![cond]\!](\rho_v, \rho_a) = false \\ [\![stmt; \textbf{while}(cond)\ stmt]\!](\rho_v, \rho_a) & otherwise \end{cases}$$
$$[\![\textbf{if}(cond)\ stmt_1\ \textbf{else}\ stmt_2]\!](\rho_v, \rho_a) =$$
$$\begin{cases} [\![stmt_1]\!](\rho_v, \rho_a) & if\ [\![cond]\!](\rho_v, \rho_a) = false \\ [\![stmt_2]\!](\rho_v, \rho_a) & otherwise \end{cases}$$

Figure 3. Semantics of Simple Programs

We use c denote a concrete program state (ρ_v, ρ_a) in subsequent paragraphs. A trace T of a CFG $\mathcal{G}$ is a potentially infinite sequence $(c_0, c_1, \ldots)$ of program states. c_0 is the initial state, and for every $i \geq 0$, we have $\mathcal{G} \vdash c_i \rightsquigarrow c_{i+1}$.

Lemma IV.10. *Let c_i denote the concrete program states immediately before executing statement n, and a_i denote the dataflow analysis information immediately before statement n. If $\mathcal{G} \vdash c_i \rightsquigarrow c_{i+1}$ and $[\![a_i]\!](c_i)$ holds, the following conditions are satisfied:*

1) *$[\![Semantics(n, a_i)]\!](c_{i+1})$ holds.*
2) *For any $S \in 2^{\mathbb{P}roperP}$, if $\mathtt{n}$ is $e_1[e_2] := e_3$ and $[\![S]\!](c_{i+1})$ holds, then $[\![Transfer(n, a_i, S)]\!](c_{i+1})$ holds.*
3) *For any $S \in 2^{\mathbb{P}roperP}$, if $\mathtt{n}$ is an index initialization statement $i := init$ and $[\![S]\!](c_{i+1})$ holds, then $[\![GenPhi(n, S)]\!](c_{i+1})$ holds.*
4) *For any $S \in 2^{\mathbb{P}roperP}$, if $\mathtt{n}$ is an index update statement $i := i + c$ and $[\![S]\!](c_{i+1})$ holds, then $[\![HandlePhi(n, a_i, S)]\!](c_{i+1})$ holds.*
5) *For any $S \in 2^{\mathbb{P}roperP}$, if $[\![S]\!](c_{i+1})$ holds, then $[\![Reduce(S)]\!](c_{i+1})$ holds.*

Proof. 1) To prove conclusion (1), we need to prove :
- if n is $lh := e$, $[\![lh]\!](c_{i+1}) = [\![e]\!](c_i) \wedge \bigwedge_{[\![\&lh \neq \&lh_i]\!](c_i) \wedge lh_i \in \mathbb{L}\mathrm{H}} [\![lh_i]\!](c_{i+1}) = [\![lh_i]\!](c_i)$
- if n is *cond*, and
  - c_{i+1} are concrete program states immediately before the *true* branch, $[\![cond]\!](c_{i+1}) \wedge \bigwedge_{lh_i \in \mathbb{L}\mathrm{H}} [\![lh_i]\!](c_{i+1}) = [\![lh_i]\!](c_i)$
  - c_{i+1} are concrete program states immediately before the *false* branch, $[\![\neg cond]\!](c_{i+1}) \wedge \bigwedge_{lh_i \in \mathbb{L}\mathrm{H}} [\![lh_i]\!](c_{i+1}) = [\![lh_i]\!](c_i)$

These conclusions follow directly from the semantics of statement (described in Fig. 3).

2) To prove conclusion (2), we need to prove :

$$[\![S \cup \{p \mid p \in a_i \wedge p \text{ is a } UQProp \wedge \&e_1[e_2] \notin \mathbb{M}(p)\}]\!](c_{i+1})$$

Since $[\![S]\!](c_{i+1})$ holds, we just need to prove: $[\![\{p \mid p \in a_i \wedge p \text{ is a } UQProp \wedge \&e_1[e_2] \notin \mathbb{M}(p)\}]\!](c_{i+1})$. $[\![a_i]\!](c_i) \wedge p \in a_i \Rightarrow [\![p]\!](c_i)$. $(\bigwedge a_i \Rightarrow \&e_1[e_2] \notin \mathbb{M}(p)) \wedge [\![a_i]\!](c_i)$ implies $[\![\&e_1[e_2] \notin \mathbb{M}(p)]\!](c_i)$. $[\![\&e_1[e_2] \notin \mathbb{M}(p)]\!](c_i)$ implies that all values of memory units in $\mathbb{M}(p)$ remain unchanged. Thus, $[\![p]\!](c_{i+1})$ holds.

3) To prove conclusion (3), we need to prove $[\![\{\forall k.k \in [init, c, i), \phi\} \cup S]\!](c_{i+1})$. Since $[\![S]\!](c_{i+1})$ holds, we just need to prove $[\![\forall k.k \in [init, c, i), \phi]\!](c_{i+1})$. Since n is $i := init$ and i does not occur in $init$, $[\![i = init]\!](c_{i+1})$ holds. $[\![i = init]\!](c_{i+1}) \Rightarrow [\![[init, step, i) = \emptyset]\!](c_{i+1}) \Rightarrow [\![\forall k.k \in [init, step, i), \phi]\!](c_{i+1})$.

4) To prove conclusion (4), we just need to prove that, if condition (4) or (5) holds, then $[\![\{\forall x.x \in [init_i, c, i), \psi(\cdots, e_1[f_1(x)], \cdots)\}]\!](c_{i+1})$ holds. Notice that $\psi(\cdots, e_1[f_1(x)], \cdots)$ does not contain i. Let $\psi(k)$ be abbreviation of $\psi(\cdots, e_1[f_1(k)], \cdots)$.
   - if condition (4) holds, $(\bigwedge a_i \Rightarrow (4)) \wedge [\![a_i]\!](c_i)$ implies $[\![(4)]\!](c_i)$. $[\![(4)]\!](c_i) \Rightarrow [\![i = init_i \wedge \psi(i)]\!](c_i)$. Since n is $i := i + c$ and $\psi(x)$ does not contain i and $[\![i = init_i]\!](c_i)$, then $[\![\psi(init_i)]\!](c_{i+1})$ and $[\![i = init_i + c]\!](c_{i+1})$. $[\![\psi(init_i)]\!](c_{i+1})$ and $[\![i = init_i + c]\!](c_{i+1})$ implies $[\![\forall x.x \in [init_i, c, i), \psi(x)]\!](c_{i+1})$.
   - if condition (5) holds, $(\bigwedge a_i \Rightarrow (5)) \wedge [\![a_i]\!](c_i)$ implies $[\![(5)]\!](c_i)$. $[\![(5)]\!](c_i) \Rightarrow [\![(\forall x.x \in [init_i, c, i), \psi_1(x)) \wedge \psi_2(i)]\!](c_i)$.
     $\psi(x) = \psi_1(x) \overline{\sqcap} \psi_2(x)$ implies $(\psi_2(x) \Rightarrow \psi(x) \wedge (\psi_1(x) \Rightarrow \psi(x))$. Thus, $\forall x.x \in [init_i, c, i), \psi_1(x) \Rightarrow \forall x.x \in [init_i, c, i), \psi(x)$.
     $[\![\forall x.x \in [init_i, c, i), \psi_1(x) \wedge \psi_2(i)]\!](c_i)$ implies $[\![\forall x.x \in [init_i, c, i), \psi(x) \wedge \psi(i)]\!](c_i)$.
     Since $[\![i]\!](c_{i+1}) = [\![i]\!](c_i) + c$ and $\psi(x)$ does not contain i, $[\![\psi(i)]\!](c_i) \Rightarrow [\![\psi(i - c)]\!](c_{i+1})$, $[\![\forall x.x \in [init_i, c, i), \psi(x)]\!](c_i) \Rightarrow [\![\forall x.x \in [init_i, c, i - c), \psi(x)]\!](c_{i+1})$.
     $[\![\forall x.x \in [init_i, c, i - c), \psi(x)]\!](c_{i+1})$ and $[\![\psi(i - c)]\!](c_{i+1})$ implies $[\![\forall x.x \in [init_i, c, i), \psi(x)]\!](c_{i+1})$.

5) $[\![S]\!](c_{i+1}) \Rightarrow [\![Reduce(S)]\!](c_{i+1})$. Thus, the conclusion (5) holds.

□

Theorem IV.4. *Let c_i denote the concrete program states immediately before executing statement n, and a_i denote the dataflow analysis information immediately before statement n. For the flow function F_n, if all propagated rules are sound and $\mathcal{G} \vdash c_i \rightsquigarrow c_{i+1}$ and $[\![a_i]\!](c_i)$ holds, $[\![F_n(a_i)]\!](c_{i+1})$ holds.*

Proof. Soundness of propagated rules means if $[\![Semantics(n, a_i)]\!](c_i \cup c_{i+1})$ holds, $[\![Propagated(a_i \cup Semantics(n, a_i))]\!](c_{i+1})$ holds. Here we use $(c_i \cup c_{i+1})$ since there are pre-state quantities (unprimed variables) and post-state quantities (unprimed variables) in $Semantics(n, a_i)$. However, there are not pre-state quantities in $Propagated(a_i \cup Semantics(n, a_i))$. According Lemma IV.10 and the soundness of propagated rules, $[\![F_n(a_i)]\!](c_{i+1})$ holds. □

Theorem IV.5. *For all initial states c_0, $[\![BI]\!](c_0)$ holds. (* BI *is the data flow values before the Start statement)*

Proof. In array properties analysis, BI is $\emptyset$. Thus, for all initial states c_0, $[\![BI]\!](c_0)$ holds. □

Theorem IV.6. *For any* $L_1, L_2 \in L_{\mathcal{G}}$, $\bigwedge L_1 \vee \bigwedge L_2 \Rightarrow \bigwedge(L_1 \sqcap_{\mathcal{G}} L_2)$

Proof. According to distributivity of $\vee$ over $\wedge$, $\bigwedge L_1 \vee \bigwedge L_2$ can be converted into an equivalent formula that is in CNF: $\bigwedge_{p_1 \in L_1 \wedge p_2 \in L_2}(p_1 \vee p_2)$. If $p_i \in L_1, p_j \in L_2$, and $p_i \overline{\sqcap} p_j$ is in $L_1 \sqcap_{\mathcal{G}} L_2$, according to Corollary IV.3, $p_i \vee p_j \Rightarrow p_i \overline{\sqcap} p_j$. Thus, we have $\bigwedge L_1 \vee \bigwedge L_2 \Rightarrow \bigwedge(L_1 \sqcap_{\mathcal{G}} L_2)$. □

Therefore, after the array properties analysis terminates, the ultimate data flow values are sound.

G. Flexibility of Method

The flexibility of our method is reflected by the abstract domain $L_{\mathcal{G}}$ and propagated rules. Actually, for a CFG $\mathcal{G}$, if a and b occur in $\mathcal{G}$, but $a+b$ does not occur in $\mathcal{G}$, we can not get any properties of $a+b$. However, we can extend the abstract domain to get more properties. For example, $\mathbb{E}xpr'$=$\mathbb{E}xpr \cup \{e_1 + e_2 \mid e_1, e_2 \in \mathbb{E}xpr\}$. We also could cut the abstract domain to make a faster convergence rate.

Similar to the abstract domain, we can extend propagated rules or reduce them. For example, at first we process *Summation starts* program (Program 5), our method can not get any universally quantified properties.

Program 5. Summation starts

```
1  negative_sum = 0;
2  positive_sum = 0;
3  for (i = 0; i < MAXSIZE; i++) {
4      if (array[i] < 0) {
5          negative_sum = negative_sum + array[i];
6      }
7      else {
8          positive_sum = positive_sum + array[i];
9      }
10 }
```

However, if we add two additional rules (Actually, since **propagated rules constraints**, we need to add more than two rules).

$$e_1 = e_2 + e_3 \wedge e_2 \geq 0 \wedge e_3 \geq 0 \Rightarrow e_1 \geq 0 \wedge e_1 \geq e_2 \wedge e_1 \geq e_3$$
$$e_1 = e_2 + e_3 \wedge e_2 < 0 \wedge e_3 < 0 \Rightarrow e_1 < 0 \wedge e_1 < e_2 \wedge e_1 < e_3$$

we can get the following properties at exit of the program:

$$\forall k.k \in [0, 1, MAXSIZE), A[k] \leq positive\_sum$$
$$\forall k.k \in [0, 1, MAXSIZE), A[k] \geq negative\_sum$$

V. Implementation and Experiments

The method has been implemented using clang [1] and Z3 [9]. In the prototype tool, Z3 is mainly used to check whether two memory units are the same. For example $\&A[i]$ and $\&A[j]$, we need to check whether i is equal to j in current state. Moreover, we use Z3 to check whether universally quantified properties are still valid after a statement.

A. Analysis of Some Examples

We give now the results of this analysis on some examples: a version of the famous Find program used for segmenting arrays in QuickSort, "arrayPartCopy" program, and two 2-dimensional array programs, they have same function, but one (Program 3) is more complicated. Note that, in our tool, we represent intervals as properties. For example, we represent $\forall k.k \in [0, 2, i), A[k] > x$ as $\forall k.0 \leq k < i \wedge i\%2 = 0 \Rightarrow A[k] > x$.

Results for "arrayPartCopy": At the end of the program, our tool automatically discovers the following properties:

- $i \geq size$
- $\forall k.0 \leq k < i \wedge k\%2 = 0 \Rightarrow A[k] = B[k]$
- $\forall k.0 \leq k < size \wedge k\%2 = 0 \Rightarrow A[k] = B[k]$

Results for "Find": "Find" program [16] is used for segmenting an array according to its first element, and used at each step of the QuickSort. At the end of the program, our tool automatically discovers the following properties:

- $i > j \wedge A[i-1] = x$
- $\forall k.1 \leq k < i \Rightarrow A[k-1] < x$
- $\forall k.1 \leq k < j \Rightarrow A[k-1] < x$
- $\forall k.i \leq k \leq size - 1 \Rightarrow A[k] >= x$

Results for "twoDimArrayCheck": "twoDimArrayCheck" program checks whether there exists a two-dimensional array element who is equal to 0. We give two versions of "twoDimArrayCheck" program. One (Program 3) is more complicated. At the end of the both programs, our tool automatically discovers the following properties for both versions:

- $\forall k_1.0 \leq k_1 < i \Rightarrow (\forall k_2.0 \leq k_2 < col \Rightarrow A[k_1][k_2] \neq 0)$

Besides the four examples, we applied our tool to automatically generate invariants for other examples, including: "partition" copies the less than and greater than or equal x elements of a source array into two different arrays; insertion sort algorithms (we were able to discover the inductive invariant of the inner loop); "ArrayMax" program, a version of the "firstNotNull"; "ArrayCopy", including 1-dimensional, 2-dimensional and 3-dimensional array program; "ArrayCheck", including 1-dimensional, 2-dimensional and 3-dimensional array program. For more tool details, please visit the web page of this tool and examples: *https://github.com/libin049/InvariantSynthesisForArray_C*.

Performances: Experiments were run on a 2.4 GHz Intel processor with 4 GB of RAM. Table III shows the performances result.

Table III shows that some small 1-dimensional array examples take less than one second to be analyzed. But multidimensional array examples require more time to be analyzed. For multidimensional array examples, tool generates a lot of universal quantifier properties. It takes a lot of time to check whether the universally quantified

properties are still valid after a statement, which requires Z3.

B. Analysis of array-examples benchmark of SV-COMP

Competition on Software Verification (SV-COMP)[3] provides an array program benchmark *array-examples* [2]. The benchmark has to be written in GNU C or ANSI C. It has 88 files, 2299 line codes. Most of files only have one function, and all of arrays are one dimension arrays. Table IV shows the execution time, basic blocks number, universal quantifier properties number(**#U-inv**) and atomic properties number(**#A-inv**) on entry of all blocks, and number of files in which our tool finds universal quantifier properties (**#file(U-inv)**). We discard the universal quantifier properties which are correct but insignificant (e.g. $(\forall k.k \in [0,1,ret),p) \wedge ret = 0$). Table IV shows that the average time for each of files is 10 seconds, and tool finds universal quantifier properties in 74 files. Our tool generates more universal quantifier properties than atomic properties since lots of universal quantifier properties are generated by propagated rules. For example, $\forall k.k \in [0,1,i),p \wedge i \geq size \wedge x < size \wedge x > 0 \Rightarrow \forall k.k \in [0,1,size),p \wedge \forall k.k \in [0,1,x),p \wedge \forall k.k \in [x,1,i),p \wedge \forall k.k \in [x,1,size),p$, tool finally synthesizes 5 universal quantifier properties.

The *array-examples* benchmark in SV-COMP is designed to check (un)reachability. Thus, universal quantifier properties are not clearly described in specifications. However, some programs of the benchmark set are provided by the projects BOOSTER [4]. In original files, universal quantifier properties are clearly described. We compare universal quantifier properties synthesized by out tool and the universal quantifier properties specifications in *standard* benchmark [4]. BOOSTER is a software model-checker devised for verifying imperative programs with flat arrays (one-dimensional arrays).

Table III
PERFORMANCES RESULTS.

Procedure	time (s)
partition	4.27
InsertionSort(inner loop)	3.22
ArrayMax	1.19
firstNotNull	2.46
Find	9.75
arrayPartCopy	0.52
1-dim ArrayCopy	0.46
2-dim ArrayCopy	6.11
3-dim ArrayCopy	43.34
1-dim ArrayCheck	0.57
2-dim ArrayCheck	4.00
3-dim ArrayCheck	57.30

Table IV
ANALYSIS RESULT OF *array-examples*

time(s)	Δ(s)	#block	#U-inv	#U-inv /#block	#A-inv	#file(U-inv)
961	10.9	1681	9978	5.94	3438	74

There are 43 files in *standard*. However, there are only 40 universal quantifier properties specifications in files. There are 33 target universal quantifier properties synthesized by our tool. We notice that, though our tool can not synthesize target universal quantifier properties in 7 files, it synthesizes other universal quantifier properties, which are useful to derive target universal quantifier properties. For example, when we process *copyInitSum* program (Program 6), our tool synthesizes the following properties before "for" statement: $\{(\forall k.k \in [0,1,N),A[k] = B[k]),(\forall k.k \in [0,1,N),A[k] = 42),(\forall k.k \in [0,1,N),B[k] = 42)\}$. The properties are useful to derive target properties. Since $42+incr \notin \mathbb{E}xpr$, our tool can not generate the target property. If we extend the abstract domain, for example, $\mathbb{E}xpr'=\mathbb{E}xpr \cup \{e_1 + e_2 \mid e_1,e_2 \in \mathbb{E}xpr\}$, we think we can synthesize the target property.

Program 6. copyInitSum

```
1  ...
2  for ( i = 0 ; i < N ; i++ ) {
3    b[i] = b[i] + incr;
4  }
5  assert( forall (int x) :: (0 <= x && x < N)
          ==> ( b[x] == 42 + incr ) );
```

VI. RELATED WORK

Several researchers have previously investigated the problem of generating array invariants.

Array expansion [6]. This method expands the cells of the array to local variables, and fully unrolls the loops. Array expansion is precise, but in practice can only be used for arrays of small size, and is not able to handle unbounded arrays.

Array smashing [7], [12]. All the cells of an array A are subsumed by one variable a. Initially, a is given the strongest known property satisfied by all the initial values of the cells of A. $A[i] := e$ is replaced by $a \sqcup := e$. However, weak assignment can only lose information tests on individual cells do not bring any information (do not process test condition). One needs to know an initial property satisfied by all the array cells. As a consequence, the results are generally unprecise.

Array partitioning [8], [13], [15]. Array partitioning method partitions the index domain (say, [1...n]) into several symbolic intervals (e.g., $I_1 = [1...i-1]$, $I_2 = [i,i]$, $I_3 = [i+1...n]$), and associates with each subarray $A[I_k]$ a summary auxiliary variable a_k. The partitioning is done either syntactically [13], [15] or by some pre-analysis [8]. Our approach does not associate with each subarray a summary auxiliary variable (or slice variables) to process. We directly process universally quantified properties, which are highly expressive. Thus, our method is very easy to deal with multidimensional arrays properties.

Predicate abstraction. Predicate abstraction [10], [20],

[22] uses some easy syntactic heuristics to derive the predicates used for the abstraction, or provided manually by the user. Moreover, counter-example guided refinement [5] and Craig interpolants [17] propose a significant improvement. Our approach is different in that we model the array properties directly.

Under-approximations and Templates [14], [24]. Under-approximations and templates method is extremely powerful yet expensive. The common idea behind these approaches is that the user provides templates that fix the structure of potential invariants. The analysis then searches for an invariant that instantiates the template parameters. Unlike our approach, the method may require the participation of users in the process.

Theorem prover-based [19], [23]. The method uses a saturation theorem prover to generate loop invariants. The idea in [19] is to encode the changes to an array at the i-th iteration as a quantified fact and then to systematically apply resolution to derive a closed form. However, these approaches are still limited due to the underlying theorem provers.

VII. Conclusion and Future Work

We propose a way of using the abstract interpretation for discovering properties about array contents. The method summarizes an array property as a universally quantified formula, and summarizes program states as properties set. It directly treats invariant properties (including universally quantified formulas and atomic formulas) as the abstract domain, and synthesizes invariants by "iterating forward" analysis from the initial state. As a consequence, our approach does not require the participation of users and automatically discover properties (including universally quantified properties and atomic properties) of programs. We prove the convergence and soundness of approach. The method has been implemented using clang and Z3. We use it to discover non-trivial properties of examples from [13], [15], and we use it to discover multidimensional array properties. Moreover, we use our tool to discover properties on *array-examples* benchmark [2] of SV-COMP [3]. The *array-examples* benchmark has 88 files; our tool finds universal quantifier properties in 74 files.

In future work, we plan to process that a universally quantified property may be divided into several universally quantified properties, or several universally quantified properties may be synthesized to a universally quantified property. A longer term perspective would plan to process universally quantified constraints and existential quantified constraints of arrays and linked lists.

Acknowledgment

This work was supported in part by the National Natural Science Foundation of China (No.61561146394, No.91318301, No.91418204, No.61321491 and No.61472179) and National Key Basic Research Program of China (No.2014CB340703).

References

[1] http://clang.llvm.org/.
[2] https://github.com/sosy-lab/sv-benchmarks/releases/tag/svcomp16.
[3] http://sv-comp.sosy-lab.org/2016/index.php.
[4] http://www.inf.usi.ch/phd/alberti/prj/booster/.
[5] D. Beyer, T. A. Henzinger, R. Majumdar, and A. Rybalchenko. Path invariants. In *Acm Sigplan Notices*, volume 42, pages 300–309. ACM, 2007.
[6] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software. In *The Essence of Computation*, pages 85–108. Springer, 2002.
[7] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *ACM SIGPLAN Notices*, volume 38, pages 196–207. ACM, 2003.
[8] P. Cousot, R. Cousot, and F. Logozzo. A parametric segmentation functor for fully automatic and scalable array content analysis. In *ACM SIGPLAN Notices*, volume 46, pages 105–118. ACM, 2011.
[9] L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.
[10] C. Flanagan and S. Qadeer. Predicate abstraction for software verification. In *ACM SIGPLAN Notices*, volume 37, pages 191–202. ACM, 2002.
[11] D. Gopan. *Numeric program analysis techniques with applications to array analysis and library summarization*. PhD thesis, Citeseer, 2007.
[12] D. Gopan, F. DiMaio, N. Dor, T. Reps, and M. Sagiv. Numeric domains with summarized dimensions. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 512–529. Springer, 2004.
[13] D. Gopan, T. Reps, and M. Sagiv. A framework for numeric analysis of array operations. *ACM SIGPLAN Notices*, 40(1):338–350, 2005.
[14] S. Gulwani, B. McCloskey, and A. Tiwari. Lifting abstract interpreters to quantified logical domains. In *ACM SIGPLAN Notices*, volume 43, pages 235–246. ACM, 2008.
[15] N. Halbwachs and M. Péron. Discovering properties about arrays in simple programs. In *ACM SIGPLAN Notices*, volume 43, pages 339–348. ACM, 2008.
[16] C. A. Hoare. Proof of a program: Find. *Communications of the ACM*, 14(1):39–45, 1971.
[17] R. Jhala and K. L. McMillan. Array abstractions from proofs. In *Computer Aided Verification*, pages 193–206. Springer, 2007.
[18] J. B. Kam and J. D. Ullman. Monotone data flow analysis frameworks. *Acta Informatica*, 7(3):305–317, 1977.
[19] L. Kovács and A. Voronkov. Finding loop invariants for programs over arrays using a theorem prover. In *Fundamental Approaches to Software Engineering*, pages 470–485. Springer, 2009.
[20] S. K. Lahiri and R. E. Bryant. Constructing quantified invariants via predicate abstraction. In *Verification, Model Checking, and Abstract Interpretation*, pages 267–281. Springer, 2004.
[21] S. K. Lahiri and R. E. Bryant. Indexed predicate discovery for unbounded system verification. In *Computer Aided Verification*, pages 135–147. Springer, 2004.
[22] S. K. Lahiri, R. E. Bryant, and B. Cook. A symbolic approach to predicate abstraction. In *Computer Aided Verification*, pages 141–153. Springer, 2003.
[23] K. L. McMillan. Quantified invariant generation using an interpolating saturation prover. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 413–427. Springer, 2008.
[24] S. Srivastava and S. Gulwani. Program verification using templates over predicate abstraction. In *ACM Sigplan Notices*, volume 44, pages 223–234. ACM, 2009.
[25] ZhaoJianhua and LiXuandong. Scope logic: An extension to hoare logic for pointers and recursive data structures. In *Theoretical Aspects of Computing–ICTAC 2013*, pages 409–426. Springer, 2013.